compact1, compact2, compact3

java.util

# Interface Comparator<T>

**Type Parameters:**

`T` – the type of objects that may be compared by this comparator

**All Known Implementing Classes:**

`Collator`, `RuleBasedCollator`

**Functional Interface:**

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

---

```
@FunctionalInterface
public interface Comparator<T>
```

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as `sorted sets` or `sorted maps`), or to provide an ordering for collections of objects that don't have a `natural ordering`.

The ordering imposed by a comparator `c` on a set of elements `S` is said to be *consistent with equals* if and only if `c.compare(e1, e2)==0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` in `S`.

Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map). Suppose a sorted set (or sorted map) with an explicit comparator `c` is used with elements (or keys) drawn from a set `S`. If the ordering imposed by `c` on `S` is inconsistent with equals, the sorted set (or sorted map) will behave "strangely." In particular the sorted set (or sorted map) will violate the general contract for set (or map), which is defined in terms of `equals`.

For example, suppose one adds two elements `a` and `b` such that `(a.equals(b) && c.compare(a, b) != 0)` to an empty `TreeSet` with comparator `c`. The second `add` operation will return true (and the size of the tree set will increase) because `a` and `b` are not equivalent from the tree set's perspective, even though this is contrary to the specification of the `Set.add` method.

Note: It is generally a good idea for comparators to also implement `java.io.Serializable`, as they may be used as ordering methods in serializable data structures (like `TreeSet`, `TreeMap`). In order for the data structure to serialize successfully, the comparator (if provided) must implement `Serializable`.

For the mathematically inclined, the *relation* that defines the *imposed ordering* that a given comparator `c` imposes on a given set of objects `S` is:

```
{(x, y) such that c.compare(x, y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that c.compare(x, y) == 0}.
```

It follows immediately from the contract for `compare` that the quotient is an *equivalence relation* on `S`, and that the imposed ordering is a *total order* on `S`. When we say that the ordering imposed by `c` on `S` is *consistent with equals*, we mean that the quotient for the ordering is the equivalence relation defined by the objects' `equals(Object)` method(s):

```
{(x, y) such that x.equals(y)}.
```

Unlike `Comparable`, a comparator may optionally permit comparison of null arguments, while maintaining the requirements for an equivalence relation.

This interface is a member of the Java Collections Framework.

**Since:**

`1.2`

**See Also:**

`Comparable`, `Serializable`

## *Method Summary*

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

| Modifier and Type | Method and Description |
|---|---|
| int | **compare**(T o1, T o2)<br>Compares its two arguments for order. |
| static <T,U extends Comparable<? super U>> Comparator<T> | **comparing**(Function<? super T,? extends U> keyExtractor)<br>Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T,U> Comparator<T> | **comparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)<br>Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator. |
| static <T> Comparator<T> | **comparingDouble**(ToDoubleFunction<? super T> keyExtractor)<br>Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T> Comparator<T> | **comparingInt**(ToIntFunction<? super T> keyExtractor)<br>Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T> Comparator<T> | **comparingLong**(ToLongFunction<? super T> keyExtractor)<br>Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| boolean | **equals**(Object obj)<br>Indicates whether some other object is "equal to" this comparator. |
| static <T extends Comparable<? super T>> Comparator<T> | **naturalOrder**()<br>Returns a comparator that compares Comparable objects in natural order. |
| static <T> Comparator<T> | **nullsFirst**(Comparator<? super T> comparator)<br>Returns a null-friendly comparator that considers null to be less than non-null. |
| static <T> Comparator<T> | **nullsLast**(Comparator<? super T> comparator)<br>Returns a null-friendly comparator that considers null to be greater than non-null. |
| default Comparator<T> | **reversed**()<br>Returns a comparator that imposes the reverse ordering of this comparator. |
| static <T extends Comparable<? super T>> Comparator<T> | **reverseOrder**()<br>Returns a comparator that imposes the reverse of the *natural ordering*. |
| default Comparator<T> | **thenComparing**(Comparator<? super T> other)<br>Returns a lexicographic-order comparator with another comparator. |
| default <U extends Comparable<? super U>> Comparator<T> | **thenComparing**(Function<? super T,? extends U> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a Comparable sort key. |
| default <U> Comparator<T> | **thenComparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)<br>Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator. |
| default Comparator<T> | **thenComparingDouble**(ToDoubleFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a double sort key. |
| default Comparator<T> | **thenComparingInt**(ToIntFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a int sort key. |
| default Comparator<T> | **thenComparingLong**(ToLongFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a long sort key. |

## *Method Detail*

### compare

```
int compare(T o1,
            T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

The implementor must ensure that `sgn(compare(x, y)) == -sgn(compare(y, x))` for all `x` and `y`. (This implies that `compare(x, y)` must throw an exception if and only if `compare(y, x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `((compare(x, y)>0) && (compare(y, z)>0))` implies `compare(x, z)>0`.

Finally, the implementor must ensure that `compare(x, y)==0` implies that `sgn(compare(x, z))==sgn(compare(y, z))` for all `z`.

It is generally the case, but *not* strictly required that `(compare(x, y)==0) == (x.equals(y))`. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

**Parameters:**

`o1 - the first object to be compared.`

`o2 - the second object to be compared.`

**Returns:**

`a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.`

**Throws:**

`NullPointerException - if an argument is null and this comparator does not permit null arguments`

`ClassCastException - if the arguments' types prevent them from being compared by this comparator.`

### equals

```
boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this comparator. This method must obey the general contract of `Object.equals(Object)`. Additionally, this method can return `true` *only* if the specified object is also a comparator and it imposes the same ordering as this comparator. Thus, `comp1.equals(comp2)` implies that `sgn(comp1.compare(o1, o2))==sgn(comp2.compare(o1, o2))` for every object reference `o1` and `o2`.

Note that it is *always* safe *not* to override `Object.equals(Object)`. However, overriding this method may, in some cases, improve performance by allowing programs to determine that two distinct comparators impose the same order.

**Overrides:**

`equals in class Object`

**Parameters:**

`obj - the reference object with which to compare.`

**Returns:**

`true only if the specified object is also a comparator and it imposes the same ordering as this comparator.`

**See Also:**

`Object.equals(Object), Object.hashCode()`

### reversed

```
default Comparator<T> reversed()
```

Returns a comparator that imposes the reverse ordering of this comparator.

**Returns:**

`a comparator that imposes the reverse ordering of this comparator.`

**Since:**

`1.8`

### thenComparing

```
default Comparator<T> thenComparing(Comparator<? super T> other)
```

Returns a lexicographic-order comparator with another comparator. If this `Comparator` considers two elements equal, i.e. `compare(a, b) == 0`, `other` is used to determine the order.

The returned comparator is serializable if the specified comparator is also serializable.

**API Note:**

For example, to sort a collection of String based on the length and then case-insensitive natural ordering, the comparator can be composed using following code,

```
Comparator<String> cmp = Comparator.comparingInt(String::length)
        .thenComparing(String.CASE_INSENSITIVE_ORDER);
```

**Parameters:**

other – the other comparator to be used when this comparator compares two objects that are equal.

**Returns:**

a lexicographic-order comparator composed of this and then the other comparator

**Throws:**

NullPointerException – if the argument is null.

**Since:**

1.8

### thenComparing

```
default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor,
                                        Comparator<? super U> keyComparator)
```

Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given `Comparator`.

**Implementation Requirements:**

This default implementation behaves as if thenComparing(comparing(keyExtractor, cmp)).

**Type Parameters:**

U – the type of the sort key

**Parameters:**

keyExtractor – the function used to extract the sort key

keyComparator – the Comparator used to compare the sort key

**Returns:**

a lexicographic-order comparator composed of this comparator and then comparing on the key extracted by the keyExtractor function

**Throws:**

NullPointerException – if either argument is null.

**Since:**

1.8

**See Also:**

comparing(Function, Comparator), thenComparing(Comparator)

### thenComparing

```
default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a `Comparable` sort key.

**Implementation Requirements:**

This default implementation behaves as if thenComparing(comparing(keyExtractor)).

**Type Parameters:**

U – the type of the Comparable sort key

**Parameters:**

keyExtractor – the function used to extract the Comparable sort key

**Returns:**

a lexicographic-order comparator composed of this and then the `Comparable` sort key.

**Throws:**

`NullPointerException` – if the argument is null.

**Since:**

1.8

**See Also:**

`comparing(Function)`, `thenComparing(Comparator)`

### thenComparingInt

```
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a `int` sort key.

**Implementation Requirements:**

This default implementation behaves as if `thenComparing(comparingInt(keyExtractor))`.

**Parameters:**

`keyExtractor` – the function used to extract the integer sort key

**Returns:**

a lexicographic-order comparator composed of this and then the int sort key

**Throws:**

`NullPointerException` – if the argument is null.

**Since:**

1.8

**See Also:**

`comparingInt(ToIntFunction)`, `thenComparing(Comparator)`

### thenComparingLong

```
default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a `long` sort key.

**Implementation Requirements:**

This default implementation behaves as if `thenComparing(comparingLong(keyExtractor))`.

**Parameters:**

`keyExtractor` – the function used to extract the long sort key

**Returns:**

a lexicographic-order comparator composed of this and then the long sort key

**Throws:**

`NullPointerException` – if the argument is null.

**Since:**

1.8

**See Also:**

`comparingLong(ToLongFunction)`, `thenComparing(Comparator)`

### thenComparingDouble

```
default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)
```

Returns a lexicographic-order comparator with a function that extracts a `double` sort key.

**Implementation Requirements:**

This default implementation behaves as if `thenComparing(comparingDouble(keyExtractor))`.

**Parameters:**

`keyExtractor` – the function used to extract the double sort key

**Returns:**

a lexicographic-order comparator composed of this and then the double sort key

**Throws:**

NullPointerException – if the argument is null.

**Since:**

1.8

**See Also:**

comparingDouble(ToDoubleFunction), thenComparing(Comparator)

### reverseOrder

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Returns a comparator that imposes the reverse of the *natural ordering*.

The returned comparator is serializable and throws NullPointerException when comparing null.

**Type Parameters:**

T – the Comparable type of element to be compared

**Returns:**

a comparator that imposes the reverse of the *natural ordering* on Comparable objects.

**Since:**

1.8

**See Also:**

Comparable

### naturalOrder

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

Returns a comparator that compares Comparable objects in natural order.

The returned comparator is serializable and throws NullPointerException when comparing null.

**Type Parameters:**

T – the Comparable type of element to be compared

**Returns:**

a comparator that imposes the *natural ordering* on Comparable objects.

**Since:**

1.8

**See Also:**

Comparable

### nullsFirst

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)
```

Returns a null-friendly comparator that considers null to be less than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order. If the specified comparator is null, then the returned comparator considers all non-null values to be equal.

The returned comparator is serializable if the specified comparator is serializable.

**Type Parameters:**

T – the type of the elements to be compared

**Parameters:**

comparator – a Comparator for comparing non-null values

**Returns:**

a comparator that considers null to be less than non-null, and compares non-null objects with the supplied Comparator.

**Since:**

1.8

#### nullsLast

```
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)
```

Returns a null-friendly comparator that considers `null` to be greater than non-null. When both are `null`, they are considered equal. If both are non-null, the specified `Comparator` is used to determine the order. If the specified comparator is `null`, then the returned comparator considers all non-null values to be equal.

The returned comparator is serializable if the specified comparator is serializable.

**Type Parameters:**

T – the type of the elements to be compared

**Parameters:**

comparator – a Comparator for comparing non-null values

**Returns:**

a comparator that considers null to be greater than non-null, and compares non-null objects with the supplied Comparator.

**Since:**

1.8

#### comparing

```
static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor,
                                     Comparator<? super U> keyComparator)
```

Accepts a function that extracts a sort key from a type T, and returns a `Comparator<T>` that compares by that sort key using the specified `Comparator`.

The returned comparator is serializable if the specified function and comparator are both serializable.

**API Note:**

For example, to obtain a Comparator that compares Person objects by their last name ignoring case differences,

```
Comparator<Person> cmp = Comparator.comparing(
        Person::getLastName,
        String.CASE_INSENSITIVE_ORDER);
```

**Type Parameters:**

T – the type of element to be compared

U – the type of the sort key

**Parameters:**

keyExtractor – the function used to extract the sort key

keyComparator – the Comparator used to compare the sort key

**Returns:**

a comparator that compares by an extracted key using the specified Comparator

**Throws:**

NullPointerException – if either argument is null

**Since:**

1.8

#### comparing

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)
```

Accepts a function that extracts a `Comparable` sort key from a type T, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

**API Note:**

For example, to obtain a Comparator that compares Person objects by their last name,

```
Comparator<Person> byLastName = Comparator.comparing(Person::getLastName);
```

**Type Parameters:**

T – the type of element to be compared

U – the type of the Comparable sort key

**Parameters:**

keyExtractor – the function used to extract the Comparable sort key

**Returns:**

a comparator that compares by an extracted key

**Throws:**

NullPointerException – if the argument is null

**Since:**

1.8

### comparingInt

```
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)
```

Accepts a function that extracts an `int` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

**Type Parameters:**

T – the type of element to be compared

**Parameters:**

keyExtractor – the function used to extract the integer sort key

**Returns:**

a comparator that compares by an extracted key

**Throws:**

NullPointerException – if the argument is null

**Since:**

1.8

**See Also:**

comparing(Function)

### comparingLong

```
static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)
```

Accepts a function that extracts a `long` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

**Type Parameters:**

T – the type of element to be compared

**Parameters:**

keyExtractor – the function used to extract the long sort key

**Returns:**

a comparator that compares by an extracted key

**Throws:**

NullPointerException – if the argument is null

**Since:**

1.8

**See Also:**

comparing(Function)

### comparingDouble

```
static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)
```

Accepts a function that extracts a `double` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

**Type Parameters:**

`T` – the type of element to be compared

**Parameters:**

`keyExtractor` – the function used to extract the double sort key

**Returns:**

a comparator that compares by an extracted key

**Throws:**

`NullPointerException` – if the argument is null

**Since:**

1.8

**See Also:**

`comparing(Function)`